How to Combine PySpark DataFrames with Different Columns

Authored by

stats writer

January 2, 2026

Data manipulation is a core task in large-scale data processing, and combining datasets is frequently required. In the PySpark environment, which leverages the distributed processing power of Apache Spark, merging data typically involves the `union` operation on DataFrame objects. However, a common challenge arises when attempting to combine DataFrames that possess different underlying structures or schemas. The standard `union()` method expects schemas to be identical in both the number of columns and their corresponding data types, demanding strict positional alignment.

When schemas diverge--meaning one DataFrame contains columns not present in the other, or the columns are simply ordered differently--the traditional union approach fails or produces incorrect results due to misaligned data. To address this heterogeneity gracefully, PySpark introduced the `unionByName()` method. This function allows for the combination of DataFrames by matching columns based on their explicit names, rather than their sequential position. Furthermore, it incorporates mechanisms to handle missing fields, ensuring a robust and flexible merging process essential for real-world big data integration scenarios.

This guide delves into how to effectively combine two DataFrames with non-identical column sets using the specialized `unionByName()` function. We will explore the critical parameter that enables this capability, walk through a comprehensive, practical example demonstrating the setup and execution, and analyze the resultant unified DataFrame, paying close attention to how missing data is managed to maintain data integrity across the combined structure.

Understanding the Need for unionByName

The standard `union()` method in PySpark relies on positional matching. If `df1` has columns (A, B, C) and `df2` has columns (X, Y, Z), a standard union attempts to combine A with X, B with Y, and C with Z, expecting them to be of compatible types. This approach is highly efficient when dealing with homogenous data sources, such as appending daily logs from identical log files. However, when DataFrames originate from disparate sources--perhaps one containing sales data (items, price, date) and another containing customer service metrics (customer_id, issue, resolution_time)--their schemas will naturally differ.

Attempting to use the positional `union()` on heterogeneous DataFrames will result in either an error if the number of columns differs, or, more dangerously, an incorrect merge if the number of columns is the same but the names or types are mismatched positionally. For instance, if `df1` has (Name, Age) and `df2` has (Age, Location), a standard union would incorrectly attempt to merge Name with Age and Age with Location, corrupting the dataset and making subsequent analysis unreliable. The solution lies in merging based on explicit column identifiers rather than their sequence.

This is where `unionByName()` becomes indispensable. It performs a schema resolution step where

it inspects the column names of both input DataFrames and ensures that matching columns are aligned correctly, regardless of their order. For any columns present in one DataFrame but absent in the other, `unionByName()` inserts placeholders, which is the key mechanism for successful heterogeneous union operations. This mechanism is activated by setting a specific parameter, as we detail in the following section.

The Syntax and the Critical Parameter

To successfully perform a union on two PySpark DataFrames that contain different columns, we utilize `unionByName()` along with a critical configuration flag. This function is designed explicitly to handle schema variation, ensuring alignment based on column names. The core syntax for this operation is straightforward, targeting the first DataFrame and applying the method with the second DataFrame as the primary argument.

The syntax below shows the required structure for combining DataFrames `df1` and `df2`, ensuring that columns are matched by name and that the operation can proceed even if one or both DataFrames are missing certain columns present in the other:

df_union = df1.unionByName(df2, allowMissingColumns=True)

This particular example performs a union between the PySpark DataFrames named **df1** and **df2**. The power of this operation lies entirely in the second argument. By using the argument **`allowMissingColumns=True`**, we explicitly instruct the function that the set of column names between the two DataFrames is allowed to differ. When this flag is set to `True`, the resulting schema of `df_union` will be the superset of the columns from both `df1` and `df2`. If a column exists in `df1` but not `df2`, the rows originating from `df2` will have null values in that column, and vice versa. This functionality is the cornerstone of flexible distributed data integration.

unionByName in Practice: Setting Up the Scenario

To illustrate the practical application of `unionByName()`, consider a scenario where we are combining two different sets of sports statistics. The first set, represented by `df1`, focuses on general team performance metrics, while the second set, `df2`, focuses specifically on player contribution metrics that may not align perfectly with the first dataset. The goal is to consolidate all records into a single DataFrame for comprehensive analysis, without losing any data points.

We begin by initializing a Spark session and defining our first DataFrame, **df1**. This DataFrame contains columns **team**, **conference**, and **points**. This DataFrame represents a dataset where information about assists or other individual metrics is simply not tracked, defining its specific schema:

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data1 = ,
,
,
,
,
]

#define column names
columns1 =

#create DataFrame
df1 = spark.createDataFrame(data1, columns1)

#view DataFrame
df1.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
A	East	11
B	East	8
C	East	31
D	West	16
E	West	6
F	East	5
+----+----------+------+

Next, we define our second DataFrame, **df2**. This DataFrame shares the **team** column but introduces a new column, **assists**, which is not present in `df1`. Crucially, `df2` lacks the **conference** and **points** columns. This clear difference in schema highlights the necessity of using `unionByName()` for a successful merge. The common column, `team`, will serve as the alignment key for the data rows, though it's important to remember that `unionByName()` simply stacks rows, it does not join data based on common key values like a SQL JOIN operation would.

#define data
data2 = ,
,

,
,
,
]

#define column names
columns2 =

#create DataFrame
df2 = spark.createDataFrame(data2, columns2)

#view DataFrame
df2.show()

+----+-------+
|team|assists|
+----+-------+
G	4
H	8
I	11
J	5
K	2
L	4
+----+-------+

Executing the Union with Schema Tolerance

With both heterogeneous DataFrames defined, we can now execute the union operation using the specialized syntax. Because we are using `unionByName()` and setting the `allowMissingColumns` flag to `True`, PySpark will automatically construct the resulting schema by aggregating all unique column names present in either `df1` or `df2`. In this specific case, the final schema will contain `team`, `conference`, `points`, and `assists`.

The execution of the union is extremely fast due to the nature of distributed processing in Apache Spark. This operation is fundamentally a row stacking operation; it does not involve expensive data shuffling beyond potentially reordering rows during the output phase, making it highly efficient for combining large volumes of data from different sources, provided the schemas are handled correctly by the `unionByName` function. We use the following syntax to perform a union on these two DataFrames:

#perform union with df1 and df2

df_union = df1.unionByName(df2, allowMissingColumns=True)

#view final DataFrame
df_union.show()

```
+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| null|
| A| East| 8| null|
| A| East| 31| null|
| B| West| 16| null|
| B| West| 6| null|
| C| East| 5| null|
| A| null| null| 4|
| A| null| null| 8|
| A| null| null| 11|
| B| null| null| 5|
| B| null| null| 2|
| C| null| null| 4|
+----+----------+------+-------+
```

Analyzing the Resultant Schema and Null Value Handling

The final unified DataFrame, `df_union`, clearly demonstrates the successful integration of the two disparate datasets. It contains all the rows from both `df1` and `df2`, and its schema is the union of the individual schemas. The key columns resulting from the merge are `team`, `conference`, `points`, and `assists`. This confirms that `unionByName()` successfully reconciled the differing column sets.

Crucially, the operation manages missing columns gracefully by inserting Null values. Observe the first six rows, which originated from `df1`: these rows contain valid data for `conference` and `points` but lack information for `assists`. Consequently, the `assists` column for these rows is populated with `null`. Conversely, the rows originating from `df2` contain valid data for `assists` but show `null` values for the `conference` and `points` columns. This mechanism guarantees that no data is lost and that the final DataFrame maintains a consistent, unified structure across all records, despite the underlying data source differences.

It is important for data engineers to understand that these `null` values are critical placeholders. Depending on the subsequent analysis, these nulls may need to be explicitly handled--perhaps by imputing default values, filtering the data, or using functions like `fillna()`. The default behavior of

`unionByName` is to use the standard Spark `null` representation for missing data, which is vital for maintaining type compatibility across the merged columns, preventing runtime errors that would occur if the columns were simply left out.

Best Practices and Performance Considerations

While `unionByName()` offers unparalleled flexibility in combining heterogeneous DataFrames, its use mandates careful consideration of data types and performance implications. When merging columns that exist in both DataFrames, their data types must be compatible. For example, if `df1.column_A` is an integer and `df2.column_A` is a string, PySpark will typically attempt to find a common, broader type (like string) for the resulting column. It is always best practice to ensure that columns with the same name also share the same logical data type before performing the union, potentially requiring explicit casting using `.withColumn()`.

In terms of performance, `unionByName()` is generally highly efficient because it avoids the costly data shuffling operations associated with joins. It performs a simple row-stacking operation after an initial metadata check and schema reconciliation. However, the schema reconciliation step itself introduces a slight overhead compared to the standard positional `union()`. Therefore, if you know your DataFrames have identical and correctly ordered schemas, the standard `union()` remains the fastest choice. Use `unionByName()` exclusively when dealing with schema differences or when column order cannot be guaranteed.

Another crucial best practice is to always define a clear, consolidated schema expectation post-union. Understanding exactly which columns will contain null values is key to preventing errors in downstream processing. If missing columns are numerous, it might indicate that the datasets should be joined (if they share a key) rather than unioned (if they represent separate observations), highlighting the importance of choosing the correct data integration strategy based on the business logic of the data itself.

Conclusion and Further Resources

The ability to handle DataFrames with differing schemas is a fundamental requirement in advanced PySpark development. By leveraging the `unionByName()` method combined with the `allowMissingColumns=True` parameter, developers can reliably consolidate heterogeneous datasets into a single, cohesive DataFrame. This approach ensures that data is aligned correctly by name, preserving data integrity and simplifying subsequent analytical tasks, even when the source data structures are inconsistent.

The key takeaway is that `unionByName()` manages complexity by automatically generating a superset schema and backfilling missing data points with nulls. This functionality is essential for large organizations integrating data from various operational systems where schema changes are

frequent or structures were never standardized. Mastering this technique is vital for anyone working on scalable ETL (Extract, Transform, Load) pipelines using distributed technologies like Apache Spark.

For detailed technical specifications, including advanced parameters and type coercion behavior, always refer to the official documentation. The following resource provides comprehensive information on the function discussed:

Note: You can find the complete documentation for the PySpark **unionByName** function here.

Related PySpark Data Manipulation Techniques

While unioning is critical for row aggregation, PySpark offers numerous other methods for data integration and manipulation. Understanding the difference between union, join, and set operations (like intersect and except) is crucial for efficient data engineering.

Joins (`join()`): Used to combine data horizontally based on a specific key or condition, suitable for enriching records from one DataFrame with corresponding fields from another. This differs fundamentally from `unionByName()`, which stacks data vertically.

Set Operations (`intersect()`, `exceptAll()`): These are used when you need to find common rows or unique rows between DataFrames, respectively. These operations typically require identical schemas, similar to the standard `union()`.

The following tutorials explain how to perform other common tasks in PySpark:

How to use window functions for advanced aggregations.

Techniques for optimizing shuffle operations in Spark SQL.

Handling schema evolution using Parquet and Delta Lake formats.